MPI-FAUN: An MPI-Based Framework for Alternating-Updating Nonnegative Matrix Factorization

Ramakrishnan Kannan, kannanr@ornl.gov, Oak Ridge National Laboratory, TN
Grey Ballard, ballard@wfu.edu, Wake Forest University, NC
Haesun Park, hpark@cc.gatech.edu, Georgia Institute of Technology, GA

Abstract—Non-negative matrix factorization (NMF) is the problem of determining two non-negative low rank factors $\mathbf{W}$ and $\mathbf{H}$, for the given input matrix $\mathbf{A}$, such that $\mathbf{A} \approx \mathbf{WH}$. NMF is a useful tool for many applications in different domains such as topic modeling in text mining, background separation in video analysis, and community detection in social networks. Despite its popularity in the data mining community, there is a lack of efficient parallel algorithms to solve the problem for big data sets. The main contribution of this work is a new, high-performance parallel computational framework for a broad class of NMF algorithms that iteratively solves alternating non-negative least squares (NLS) subproblems for $\mathbf{W}$ and $\mathbf{H}$. It maintains the data and factor matrices in memory (distributed across processors), uses MPI for interprocessor communication, and, in the dense case, provably minimizes communication costs (under mild assumptions). The framework is flexible and able to leverage a variety of NMF and NLS algorithms, including Multiplicative Update, Hierarchical Alternating Least Squares, and Block Principal Pivoting. Our implementation allows us to benchmark and compare different algorithms on massive dense and sparse data matrices of size that spans from few hundreds of millions to billions. We demonstrate the scalability of our algorithm and compare it with baseline implementations, showing significant performance improvements. The code and the datasets used for conducting the experiments are available online.

Index Terms—HPC, NMF, MPI, 2D

1 Introduction

Non-negative Matrix Factorization (NMF) is the problem of finding two low rank factors $\mathbf{W} \in \mathbb{R}_+^{m \times k}$ and $\mathbf{H} \in \mathbb{R}_+^{k \times n}$ for a given input matrix $\mathbf{A} \in \mathbb{R}_+^{m \times n}$, such that $\mathbf{A} \approx \mathbf{WH}$. Here, $\mathbb{R}_+^{m \times n}$ denotes the set of $m \times n$ matrices with non-negative real values. Formally, the NMF problem [1] can be defined as

$$\min_{\mathbf{W} \geqslant 0, \mathbf{H} \geqslant 0} \|\mathbf{A} - \mathbf{WH}\|_F, \tag{1}$$

where $\|\mathbf{X}\|_F = (\sum_{ij} x_{ij}^2)^{1/2}$ is the Frobenius norm.

NMF is widely used in data mining and machine learning as a dimension reduction and factor analysis method. It is a natural fit for many real world problems as the non-negativity is inherent in many representations of real-world data and the resulting low rank factors are expected to have a natural interpretation. The applications of NMF range from text mining [2], computer vision [3], [4], [5], and bioinformatics [6] to blind source separation [7], unsupervised clustering [8], [9] and many other areas. In the typical case, $k \ll \min(m,n)$; for problems today, m and n can be on the order of millions or more, and k is on the order of few tens to thousands.

There is a vast literature on algorithms for NMF and their convergence properties [10]. The commonly adopted NMF algorithms are – (i) Multiplicative Update (**MU**) [1] (ii) Hierarchical Alternating Least Squares (**HALS**) [7], [11] (iii) NMF based on Alternating Nonnegative Least Squares and Block Principal Pivoting (**ABPP**) [12], and (iv) Stochastic Gradient Descent (SGD) Updates [13]. Most of the algorithms in NMF literature are based on alternately optimizing each of the low rank factors $\mathbf{W}$ and $\mathbf{H}$ while keeping the other fixed, in which case each subproblem is a constrained convex optimization problem. Subproblems can then be solved using standard optimization techniques such as projected gradient or interior point method; a detailed survey for solving such problems can be found in [14], [10]. In this paper, our implementation uses either **ABPP**, **MU**, or **HALS**. But our parallel framework is extensible to other algorithms (e.g., [15], [16]) as-is or with a few modifications, as long as they fit an alternating-updating framework (defined in Section 4).

With the advent of large scale internet data and interest in Big Data, researchers have started studying scalability of many foundational machine learning algorithms. To illustrate the dimension of matrices commonly used in the machine learning community, we present a few examples. Nowadays the adjacency matrix of a billion-node social network is common. In the matrix representation of a video data, every frame contains three matrices for each RGB color, which is reshaped into a column. Thus in the case of a 4K video, every frame will take approximately 27 million rows (4096 row pixels x 2196 column pixels x 3 colors). Similarly, the popular representation of documents in text mining is a bag-of-words matrix, where the rows are the dictionary and the columns are the documents (e.g., webpages). Each entry A_{ij} in the bag-of-words matrix is generally the frequency count of the word i in the document j. Typically with the explosion of the new terms in social media, the number of words spans to millions. To handle such high-dimensional matrices, it is important to study

low-rank approximation methods in a data-distributed and parallel computing environment.

In this work, we present an efficient algorithm and implementation using tools from the field of High-Performance Computing (HPC). We maintain data in memory (distributed across processors), take advantage of optimized libraries like BLAS and LAPACK for local computational routines, and use the Message Passing Interface (MPI) standard to organize interprocessor communication. Furthermore, the current hardware trend is that available parallelism (and therefore aggregate computational rate) is increasing much more quickly than improvements in network bandwidth and latency, which implies that the relative cost of communication (compared to computation) is increasing. To address this challenge, we analyze algorithms in terms of both their computation and communication costs. In particular, we prove in Section 5.2 that in the case of dense input and under a mild assumption, our proposed algorithm minimizes the amount of data communicated between processors to within a constant factor of the lower bound.

We call our implementation MPI-FAUN, an MPI-based Framework for Alternating-Updating Nonnegative matrix factorization algorithms. A key attribute of our framework is that the efficiency does not require a loss of generality of NMF algorithms. Our central observation is that most NMF algorithms, in particular those that alternate between updating each factor matrix, consist of two main tasks: (a) performing matrix multiplications and (b) solving Non-negative Least Squares (NLS) subproblems, either approximately or exactly. More importantly, NMF algorithms tend to perform the same matrix multiplications, differing only in how they solve NLS subproblems, and the matrix multiplications often dominate the running time of the algorithms. Our framework is designed to perform the matrix multiplications efficiently and organize the data so that the NLS subproblems can be solved independently in parallel, leveraging any of a number of possible methods. We explore the overall efficiency of the framework and compare three different NMF methods in Section 6, performing convergence, scalability, and parameter-tuning experiments on over 1500 processors.

Dataset	Type	Matrix size	NMF Time
Video	Dense	1 Million x 13,824	5.73 seconds
Stack Exchange	Sparse	627,047 x 12 Million	67 seconds
Webbase-2001	Sparse	118 Million x 118 Million	25 minutes

TABLE 1: MPI-FAUN on large real-world datasets. Reported time is for 30 iterations on 1536 processors with a low rank of 50.

With our framework, we are able to explore several large-scale synthetic and real-world data sets, some dense and some sparse. In Table 1, we present the NMF computation wall clock time on some very large real world datasets. We describe the results of the computation in Section 6, showing the range of application of NMF and the ability of our framework to scale to large data sets.

A preliminary version of this work has already appeared as a conference paper [17]. While the focus of the previous work was parallel performance of **ABPP** (Alternating Nonnegative Least Squares and Block Principal Pivoting), the goal of this paper is to explore more data analytic questions. In particular, the new contributions of this paper include (1) implementing a software framework to compare **ABPP** with **MU** (Multiplicative Update) and **HALS** (Hierarchical Alternating Least Squares) for large scale data sets, (2) benchmarking on a data analysis cluster and scaling up to over 1500 processors, and (3) providing an interpretation of results for real-world data sets. We provide a detailed comparison with other related work, including MapReduce implementations of NMF, in Section 3.

$\mathbf{A}$	Input matrix
$\mathbf{W}$	Left low rank factor
$\mathbf{H}$	Right low rank factor
m	Number of rows of input matrix
n	Number of columns of input matrix
k	Low rank
$\mathbf{M}_i$	ith row block of matrix $\mathbf{M}$
$\mathbf{M}^i$	ith column block of matrix $\mathbf{M}$
$\mathbf{M}_{ij}$	(i,j)th subblock of $\mathbf{M}$
p	Number of parallel processes
p_r	Number of rows in processor grid
p_c	Number of columns in processor grid

TABLE 2: Notation

Our main contribution is a new, high-performance parallel computational framework for a broad class of NMF algorithms. The framework is efficient, scalable, flexible, and demonstrated to be effective for large-scale dense and sparse matrices. Based on our survey and knowledge, we are the fastest NMF implementation available in the literature. The code and the datasets used for conducting the experiments can be downloaded from https://github.com/ramkikannan/nmflibrary.

2 Preliminaries

2.1 Notation

Table 2 summarizes the notation we use throughout this paper. We use *upper case* letters for matrices and *lower case* letters for vectors. We use both subscripts and superscripts for sub-blocks of matrices. For example, $\mathbf{A}_i$ is the ith row block of matrix $\mathbf{A}$, and $\mathbf{A}^i$ is the ith column block. Likewise, $\mathbf{a}_i$ is the ith row of $\mathbf{A}$, and $\mathbf{a}^i$ is the ith column. We use m and n to denote the numbers of rows and columns of $\mathbf{A}$, respectively, and we assume without loss of generality $m \geqslant n$ throughout.

2.2 Communication model

To analyze our algorithms, we use the α-β-γ model of distributed-memory parallel computation. In this model, interprocessor communication occurs in the form of messages sent between two processors across a bidirectional link (we assume a fully connected network). We model the cost of a message of size n words as $\alpha+n\beta$, where α is the per-message latency cost and β is the per-word bandwidth cost. Each processor can compute floating point operations (flops) on data that resides in its local memory; γ is the per-flop computation cost. With this communication model, we can predict the performance of an algorithm in terms of the number of flops it performs as well as the number of words and messages it communicates. For simplicity, we will ignore the possibilities of overlapping computation with communication in our analysis. For more details on the α-β-γ model, see [18], [19].

2.3 MPI collectives

Point-to-point messages can be organized into collective communication operations that involve more than two processors. MPI provides an interface to the most commonly used collectives like broadcast, reduce, and gather, as the algorithms for these collectives can be optimized for particular network topologies and processor characteristics. For a concise description of the most common collectives, see [19, Figure 1]. The algorithms we consider

use the all-gather, reduce-scatter, and all-reduce collectives, so we review them here, along with their costs. Our analysis assumes optimal collective algorithms are used (see [18], [19]), though our implementation relies on the underlying MPI implementation.

At the start of an all-gather collective, each of p processors owns data of size n/p. After the all-gather, each processor owns a copy of the entire data of size n. The cost of an all-gather is $\alpha \cdot \log p + \beta \cdot \frac{p-1}{p} n$. At the start of a reduce-scatter collective, each processor owns data of size n. After the reduce-scatter, each processor owns a subset of the sum over all data, which is of size n/p. This single collective is a more efficient way of implementing a reduce followed by a scatter. (Note that the reduction can be computed with other associative operators besides addition.) The cost of an reduce-scatter is $\alpha \cdot \log p + (\beta+\gamma) \cdot \frac{p-1}{p} n$. At the start of an all-reduce collective, each processor owns data of size n. After the all-reduce, each processor owns a copy of the sum over all data, which is also of size n. The cost of an all-reduce is $2\alpha \cdot \log p + (2\beta+\gamma) \cdot \frac{p-1}{p} n$. Note that the costs of each of the collectives are zero when $p = 1$.

3 Related Work

In the data mining and machine learning literature there is an overlap between low rank approximations and matrix factorizations due to the nature of applications. Despite its name, non-negative matrix "factorization" is really a low rank approximation. Recently there is a growing interest in collaborative filtering based recommender systems. One of the popular techniques for collaborative filtering is matrix factorization, often with nonnegativity constraints, and its implementation is widely available in many off-the-shelf distributed machine learning libraries such as GraphLab [20], MLLib [21], and many others [22], [23] as well. However, we would like to clarify that collaborative filtering using matrix factorization is a different problem than NMF: in the case of collaborative filtering, non-nonzeros in the matrix are considered to be missing entries, while in the case of NMF, non-nonzeros in the matrix correspond to true zero values.

There are several recent distributed NMF algorithms in the literature [24], [25], [26], [27]. Liu et al. propose running Multiplicative Update (MU) for KL divergence, squared loss, and "exponential" loss functions [27]. Matrix multiplication, element-wise multiplication, and element-wise division are the building blocks of the MU algorithm. The authors discuss performing these matrix operations effectively in Hadoop for sparse matrices. Using similar approaches, Liao et al. implement an open source Hadoop-based MU algorithm and study its scalability on large-scale biological data sets [24]. Also, Yin, Gao, and Zhang present a scalable NMF that can perform frequent updates, which aim to use the most recently updated data [26]. Similarly Faloutsos et al. propose a distributed, scalable method for decomposing matrices, tensors, and coupled data sets through stochastic gradient descent on a variety of objective functions [25]. The authors also provide an implementation that can enforce non-negative constraints on the factor matrices. All of these works use Hadoop to implement their algorithms.

We emphasize that our MPI-based approach has several advantages over Hadoop-based approaches:

- efficiency – our approach maintains data in memory, never communicating the data matrix, while Hadoop-based approaches must read/write data to/from disk and involves global shuffles of data matrix entries;
- generality – our approach is well-designed for both dense and sparse data matrices, whereas Hadoop-based approaches generally require sparse inputs;
- privacy – our approach allows processors to collaborate on computing an approximation without ever sharing their local input data (important for applications involving sensitive data, such as electronic health records), while Hadoop requires the user to relinquish control of data placement.

We note that Spark [28] is a popular big-data processing infrastructure that is generally more efficient for iterative algorithms such as NMF than Hadoop, as it maintains data in memory and avoids file system I/O. Even with a Spark implementation of previously proposed Hadoop-based NMF algorithm, we expect performance to suffer from expensive communication of input matrix entries, and Spark will not overcome the shortcomings of generality and privacy of the previous algorithms. Although Spark has collaborative filtering libraries such as MLlib [21], which use matrix factorization and can impose non-negativity constraints, none of them implement pure NMF, and so we do not have a direct comparison against NMF running on Spark. As mentioned above, the problem of collaborative filtering is different from NMF, and therefore different computations are performed at each iteration.

Fairbanks et al. [32] present a parallel NMF algorithm designed for multicore machines. To demonstrate the importance of minimizing communication, we consider this approach to parallelizing an alternating-updating NMF algorithm in distributed memory (see Section 5.1). While this naive algorithm exploits the natural parallelism available within the alternating iterations (the fact that rows of $\mathbf{W}$ and columns of $\mathbf{H}$ can be computed independently), it performs more communication than necessary to set up the independent problems. We compare the performance of this algorithm with our proposed approach to demonstrate the importance of designing algorithms to minimize communication; that is, simply parallelizing the computation is not sufficient for satisfactory performance and parallel scalability.

Apart from distributed NMF algorithms using Hadoop and multicores, there are also implementations of the MU algorithm in a distributed memory setting using X10 [33] and on a GPU [34].

4 Alternating-Updating NMF Algorithms

We define Alternating-Updating NMF algorithms as those that (1) alternate between updating $\mathbf{W}$ for a given $\mathbf{H}$ and updating $\mathbf{H}$ for a given $\mathbf{W}$ and (2) use the Gram matrix associated with the fixed factor matrix and the product of the input data matrix $\mathbf{A}$ with the fixed factor matrix. We show the structure of the framework in Algorithm 1.

Algorithm 1 $[\mathbf{W},\mathbf{H}] = \text{AU-NMF}(A,k)$

Require: $\mathbf{A}$ is an $m \times n$ matrix, k is the approximation rank
1: Initialize $\mathbf{H}$ with a non-negative matrix in $\mathbb{R}_+^{n \times k}$.
2: **while** stopping criteria not satisfied **do**
3: Update $\mathbf{W}$ using $\mathbf{H}\mathbf{H}^T$ and $\mathbf{A}\mathbf{H}^T$
4: Update $\mathbf{H}$ using $\mathbf{W}^T\mathbf{W}$ and $\mathbf{W}^T\mathbf{A}$
5: **end while**

The specifics of lines 3 and 4 depend on the NMF algorithm, and we refer to the computation associated with these lines as the Local Update Computations (LUC), as they will not affect the parallelization schemes we define in Section 5.2. Because these computations are performed locally, we use a function $F(m,n,k)$ to

denote the number of flops required for each algorithm's LUC (and we do not consider communication costs). Note that $F(m,n,k)$ does not include the cost of computing $\mathbf{HH}^T$, $\mathbf{W}^T\mathbf{W}$, $\mathbf{W}^T\mathbf{A}$, or $\mathbf{AH}^T$.

We note that AU-NMF is very similar to a two-block, block coordinate descent (BCD) framework, but it has a key difference. In the BCD framework where the two blocks are the unknown factors $\mathbf{W}$ and $\mathbf{H}$, we *solve* the following subproblems, which have a unique solution for a full rank $\mathbf{H}$ and $\mathbf{W}$:

$$\begin{aligned}\mathbf{W} &\leftarrow \underset{\tilde{\mathbf{W}}\geqslant 0}{\operatorname{argmin}} \left\|\mathbf{A}-\tilde{\mathbf{W}}\mathbf{H}\right\|_F, \\ \mathbf{H} &\leftarrow \underset{\tilde{\mathbf{H}}\geqslant 0}{\operatorname{argmin}} \left\|\mathbf{A}-\mathbf{W}\tilde{\mathbf{H}}\right\|_F.\end{aligned} \tag{2}$$

Since each subproblem involves nonnegative least squares, this two-block BCD method is also called the Alternating Nonnegative Least Squares (ANLS) method [10]. For example, Block Principal Pivoting (**ABPP**), discussed more in detail at Section 4.3, is one algorithm that solves these NLS subproblems. In the context of the AU-NMF algorithm, an ANLS method *maximally* reduces the overall NMF objective function value by finding the optimal solution for given $\mathbf{H}$ and $\mathbf{W}$ in lines 3 and 4 respectively.

There are other popular NMF algorithms that update the factor matrices alternatively without maximally reducing the objective function value each time, in the same sense as in ANLS. These updates do not necessarily solve each of the subproblems (2) to optimality but simply improve the overall objective function (1). Such methods include Multiplicative Update (**MU**) [1] and Hierarchical Alternating Least Squares (**HALS**) [7], which was also proposed as Rank-one Residual Iteration (RRI) [11]. To show how these methods can fit into the AU-NMF framework, we discuss them in more detail in Sections 4.1 and 4.2.

The convergence properties of these different algorithms are discussed in detail by Kim, He and Park [10]. We emphasize here that both **MU** and **HALS** require computing Gram matrices and matrix products of the input matrix and each factor matrix. Therefore, if the update ordering follows the convention of updating all of $\mathbf{W}$ followed by all of $\mathbf{H}$, both methods fit into the AU-NMF framework. We note that both **MU** and **HALS** are defined for more general update orders, but for our purposes we constrain them to be AU-NMF algorithms.

While we focus on three NMF algorithms in this paper, we highlight that our framework is extensible to other NMF algorithms, including those based on Alternating Direction Method of Multipliers (ADMM) [35], Nesterov-based methods [36], or any other method that fits the framework of Algorithm 1.

4.1 Multiplicative Update (MU)

In the case of **MU** [1], individual entries of $\mathbf{W}$ and $\mathbf{H}$ are updated with all other entries fixed. In this case, the update rules are

$$\begin{aligned}w_{ij} &\leftarrow w_{ij}\frac{(\mathbf{AH}^T)_{ij}}{(\mathbf{WHH}^T)_{ij}}, \text{ and} \\ h_{ij} &\leftarrow h_{ij}\frac{(\mathbf{W}^T\mathbf{A})_{ij}}{(\mathbf{W}^T\mathbf{WH})_{ij}}.\end{aligned} \tag{3}$$

Instead of performing these $(m+n)k$ in an arbitrary order, if all of $\mathbf{W}$ is updated before $\mathbf{H}$ (or vice-versa), this method also follows the AU-NMF framework. After computing the Gram matrices $\mathbf{HH}^T$ and $\mathbf{W}^T\mathbf{W}$ and the products $\mathbf{AH}^T$ and $\mathbf{W}^T\mathbf{A}$, the extra cost of computing $\mathbf{W}(\mathbf{HH}^T)$ and $(\mathbf{W}^T\mathbf{W})\mathbf{H}$ is $F(m,n,k) = 2(m+n)k^2$ flops to perform updates for all entries of $\mathbf{W}$ and $\mathbf{H}$, as the other elementwise operations affect only lower-order terms. Thus, when **MU** is used, lines 3 and 4 in Algorithm 1 – and functions UpdateW and UpdateH in Algorithms 2 and 3 – implement the expressions in (3), given the previously computed matrices.

4.2 Hierarchical Alternating Least Squares (HALS)

In the case of **HALS** [7], [37], updates are performed on individual columns of $\mathbf{W}$ and rows of $\mathbf{H}$ with all other entries in the factor matrices fixed. This approach is a BCD method with $2k$ blocks, set to minimize the function

$$f(\mathbf{w}^1,\cdots,\mathbf{w}^k,\mathbf{h}_1,\cdots,\mathbf{h}_k) = \left\|\mathbf{A}-\sum_{i=1}^{k}\mathbf{w}^i\mathbf{h}_i\right\|_F, \tag{4}$$

where $\mathbf{w}^i$ is the ith column of $\mathbf{W}$ and $\mathbf{h}_i$ is the ith row of $\mathbf{H}$. The update rules [37, Algorithm 2] can be written in closed form:

$$\begin{aligned}\mathbf{w}^i &\leftarrow \left[\mathbf{w}^i + (\mathbf{AH}^T)^i - \mathbf{W}(\mathbf{HH}^T)^i\right]_+ \\ \mathbf{w}^i &\leftarrow \frac{\mathbf{w}^i}{\|\mathbf{w}^i\|}, \text{ and} \\ \mathbf{h}_i &\leftarrow \left[\mathbf{h}_i + (\mathbf{W}^T\mathbf{A})_i - (\mathbf{W}^T\mathbf{W})_i\mathbf{H}\right]_+.\end{aligned} \tag{5}$$

Note that the columns of $\mathbf{W}$ and rows of $\mathbf{H}$ are updated in order, so that the most up-to-date values are always used, and these $2k$ updates can be done in an arbitrary order. However, if all the $\mathbf{W}$ updates are done before $\mathbf{H}$ (or vice-versa), the method falls into the AU-NMF framework. After computing the matrices $\mathbf{HH}^T$, $\mathbf{AH}^T$, $\mathbf{W}^T\mathbf{W}$, and $\mathbf{W}^T\mathbf{A}$, the extra computation is $F(m,n,k) = 2(m+n)k^2$ flops for updating both $\mathbf{W}$ and $\mathbf{H}$.

Thus, when **HALS** is used, lines 3 and 4 in Algorithm 1 – and functions UpdateW and UpdateH in Algorithms 2 and 3 – implement the expressions in (5), given the previously computed matrices.

4.3 Alternating Nonnegative Least Squares with Block Principal Pivoting

Block Principal Pivoting (BPP) is an active-set-like method for solving the NLS subproblems in Eq. (2). The main subroutine of BPP is the single right-hand side NLS problem

$$\min_{\mathbf{x}\geqslant 0}\|\mathbf{Cx}-\mathbf{b}\|_2. \tag{6}$$

The Karush-Kuhn-Tucker (KKT) optimality conditions for Eq. (6) are as follows

$$\mathbf{y} = \mathbf{C}^T\mathbf{Cx}-\mathbf{C}^T\mathbf{b} \tag{7a}$$

$$\mathbf{x},\mathbf{y}\geqslant 0 \tag{7b}$$

$$x_i y_i = 0\ \forall i. \tag{7c}$$

The KKT conditions (7) states that at optimality, the support sets (i.e., the non-zero elements) of $\mathbf{x}$ and $\mathbf{y}$ are complementary to each other. Therefore, Eq. (7) is an instance of the *Linear Complementarity Problem* (LCP) which arises frequently in quadratic programming. When $k \ll \min(m, n)$, active-set and active-set-like methods are very suitable because most computations involve matrices of sizes $m\times k, n\times k$, and $k\times k$ which are small and easy to handle.

If we knew which indices correspond to nonzero values in the optimal solution, then computing the solution is an unconstrained least squares problem on these indices. In the optimal solution, call the set of indices i such that $x_i = 0$ the active set, and let the remaining indices be the passive set. The BPP algorithm works to find this final active set and passive set. It greedily swaps indices between

Algorithm 2 $[\mathbf{W},\mathbf{H}] = $ Naive-Parallel-AUNMF($\mathbf{A},k$)

Require: $\mathbf{A}$ is an $m \times n$ matrix distributed both row-wise and column-wise across p processors, k is the approximation rank
Require: Local matrices: $\mathbf{A}_i$ is $m/p \times n$, $\mathbf{A}^i$ is $m \times n/p$, $\mathbf{W}_i$ is $m/p \times k$, $\mathbf{H}^i$ is $k \times n/p$
1: p_i initializes $\mathbf{H}^i$
2: **while** stopping criteria not satisfied **do**
  /* Compute W given H */
3: collect $\mathbf{H}$ on each processor using all-gather
4: p_i computes $\mathbf{W}_i \leftarrow$ updateW($\mathbf{HH}^T$,$\mathbf{A}_i\mathbf{H}^T$)
  /* Compute H given W */
5: collect $\mathbf{W}$ on each processor using all-gather
6: p_i computes $(\mathbf{H}^i)^T \leftarrow$ updateH($\mathbf{W}^T\mathbf{W}$,$(\mathbf{W}^T\mathbf{A}^i)^T$)
7: **end while**
Ensure: $\mathbf{W},\mathbf{H} \approx \underset{\tilde{\mathbf{W}}\geqslant 0,\tilde{\mathbf{H}}\geqslant 0}{\operatorname{argmin}} \|\mathbf{A}-\tilde{\mathbf{W}}\tilde{\mathbf{H}}\|$
Ensure: $\mathbf{W}$ is an $m \times k$ matrix distributed row-wise across processors, $\mathbf{H}$ is a $k \times n$ matrix distributed column-wise across processors

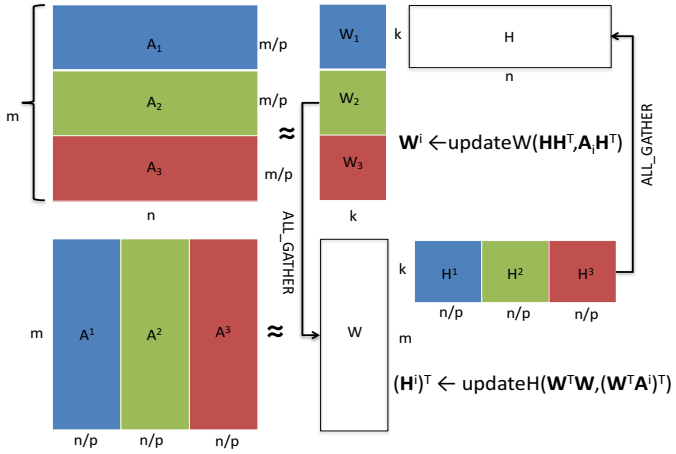


Fig. 1: Naive-Parallel-AUNMF. Both rows and columns of A are 1D distributed. The algorithm works by (all-)gathering the entire fixed factor matrix to each processor and then performing the Local Update Computations to update the variable factor matrix.

the intermediate active and passive sets until finding a partition that satisfies the KKT condition. In the partition of the optimal solution, the values of the indices that belong to the active set will take zero. The values of the indices that belong to the passive set are determined by solving the unconstrained least squares problem restricted to the passive set. Kim, He and Park [12], discuss the BPP algorithm in further detail. We use the notation

$$\mathbf{X} \leftarrow \text{SolveBPP}(\mathbf{C}^T\mathbf{C},\mathbf{C}^T\mathbf{B})$$

to define the (local) function for using BPP to solve Eq. (6) for every column of $\mathbf{X}$. We define $C_{\text{BPP}}(k,c)$ as the cost of SolveBPP, given the $k \times k$ matrix $\mathbf{C}^T\mathbf{C}$ and $k \times c$ matrix $\mathbf{C}^T\mathbf{B}$. SolveBPP mainly involves solving least squares problems over the intermediate passive sets. Our implementation uses the normal equations to solve the unconstrained least squares problems because the normal equations matrices have been pre-computed in order to check the KKT condition. However, more numerically stable methods such as QR decomposition can also be used.

Thus, when **ABPP** is used, lines 3 and 4 in Algorithm 1 – and functions UpdateW and UpdateH in Algorithms 2 and 3 – correspond to calls to SolveBPP. The number of flops involved in SolveBPP is not a closed form expression; in this case $F(m,n,k) = C_{\text{BPP}}(k,m) + C_{\text{BPP}}(k,n)$.

5 Parallel Algorithms

5.1 Naive Parallel NMF Algorithm

In this section we present a naive parallelization of NMF algorithms, which has previously appeared in the context of a shared-memory parallel platform [32]. Each NLS problem with multiple right-hand sides can be parallelized based on the observation that each right-hand side is independent from the others. For example, we can solve several instances of Eq. (6) independently for different $\mathbf{b}$ where $\mathbf{C}$ is fixed, which implies that we can optimize row blocks of $\mathbf{W}$ and column blocks of $\mathbf{H}$ in parallel.

Algorithm 2 and Figure 1 present a straightforward approach to parallelizing the independent subproblems. Let us divide $\mathbf{W}$ into row blocks $\mathbf{W}_1,\ldots,\mathbf{W}_p$ and $\mathbf{H}$ into column blocks $\mathbf{H}^1,\ldots,\mathbf{H}^p$. We then double-partition the data matrix $\mathbf{A}$ accordingly into row blocks $\mathbf{A}_1,\ldots,\mathbf{A}_p$ and column blocks $\mathbf{A}^1,\ldots,\mathbf{A}^p$ so that processor i owns both $\mathbf{A}_i$ and $\mathbf{A}^i$ (see Figure 1). With these partitions of the data and the variables, one can implement any AU-NMF algorithm in parallel, with only one communication step for each solve.

We summarize the algorithmic costs of Algorithm 2 (derived in the following subsections) in Table 3. This naive algorithm [32] has three main drawbacks: (1) it requires storing two copies of the data matrix (one in row distribution and one in column distribution) and both full factor matrices locally, (2) it does not parallelize the computation of $\mathbf{HH}^T$ and $\mathbf{W}^T\mathbf{W}$ (each processor computes it redundantly), and (3) as we will see in Section 5.2, it communicates more data than necessary.

5.1.1 Computation Cost

The computation cost of Algorithm 2 depends on the particular NMF algorithm used. Thus, the computation at line 4 consists of computing $\mathbf{A}_i\mathbf{H}^T$, $\mathbf{HH}^T$, and performing the algorithm-specific Local Update Computations for m/p rows of $\mathbf{W}$. Likewise, the computation at line 6 consists of computing $\mathbf{W}^T\mathbf{A}^i$, $\mathbf{W}^T\mathbf{W}$, and performing the Local Update Computations for n/p columns of $\mathbf{H}$. In the dense case, this amounts to $4mnk/p + (m+n)k^2 + F(m/p,n/p,k)$ flops. Note that the first term has a constant 4 to account for both $\mathbf{W}^T\mathbf{A}$ and $\mathbf{AH}^T$ and that the second term has a constant factor of 1 instead of 2 because the Gram computations ($\mathbf{HH}^T$ and $\mathbf{W}^T\mathbf{W}$) exploit symmetry of the output matrix. In the sparse case, processor i performs $2(\text{nnz}(\mathbf{A}_i)+\text{nnz}(\mathbf{A}^i))k$ flops to compute $\mathbf{A}^i\mathbf{H}^T$ and $\mathbf{W}^T\mathbf{A}_i$ instead of $4mnk/p$.

5.1.2 Communication Cost

The size of $\mathbf{W}$ is mk words, and the size of $\mathbf{H}$ is nk words. Thus, the communication cost of the all-gathers at lines 3 and 5, based on the expression given in Section 2.3 is $\alpha \cdot 2\log p + \beta \cdot (m+n)k$.

5.1.3 Memory Requirements

The local memory requirement includes storing each processor's part of matrices $\mathbf{A}$, $\mathbf{W}$, and $\mathbf{H}$. In the case of dense $\mathbf{A}$, this is $2mn/p + (m+n)k/p$ words, as $\mathbf{A}$ is stored twice; in the sparse case, processor i requires $\text{nnz}(\mathbf{A}_i) + \text{nnz}(\mathbf{A}^i)$ words for the input matrix and $(m+n)k/p$ words for the output factor matrices. Local memory is also required for storing temporary matrices $\mathbf{W}$ and $\mathbf{H}$ of size $(m+n)k$ words.

Algorithm	Flops	Words	Messages	Memory
Naive-Parallel-AUNMF	$4\frac{mnk}{p}+(m+n)k^2+F\left(\frac{m}{p},\frac{n}{p},k\right)$	$O((m+n)k)$	$O(\log p)^*$	$O\left(\frac{mn}{p}+(m+n)k\right)$
MPI-FAUN ($m/p \geqslant n$)	$4\frac{mnk}{p}+\frac{(m+n)k^2}{p}+F\left(\frac{m}{p},\frac{n}{p},k\right)$	$O(nk)$	$O(\log p)^*$	$O\left(\frac{mn}{p}+\frac{mk}{p}+nk\right)$
MPI-FAUN ($m/p<n$)	$4\frac{mnk}{p}+\frac{(m+n)k^2}{p}+F\left(\frac{m}{p},\frac{n}{p},k\right)$	$O\left(\sqrt{\frac{mnk^2}{p}}\right)$	$O(\log p)^*$	$O\left(\frac{mn}{p}+\sqrt{\frac{mnk^2}{p}}\right)$
Lower Bound	–	$\Omega\left(\min\left\{\sqrt{\frac{mnk^2}{p}},nk\right\}\right)$	$\Omega(\log p)$	$\frac{mn}{p}+\frac{(m+n)k}{p}$

TABLE 3: Leading order algorithmic costs for Naive-Parallel-AUNMF and MPI-FAUN (per iteration). Note that the computation and memory costs assume the data matrix $\mathbf{A}$ is dense, but the communication costs (words and messages) apply to both dense and sparse cases. The function $F(\cdot)$ denotes the number of flops required for the particular NMF algorithm's Local Update Computation, aside from the matrix multiplications common across AU-NMF algorithms. Note that $F(m,n,k)$ is proportional to $m+n$ and not mn, so the term in the table scales linearly with p (and not p^2) for all LUC.
*The stated latency cost assumes no communication is required in LUC; **HALS** requires $k\log p$ messages for normalization steps.

5.2 MPI-FAUN

We present our proposed algorithm, MPI-FAUN, as Algorithm 3. The main ideas of the algorithm are to (1) exploit the independence of Local Update Computations for rows of $\mathbf{W}$ and columns of $\mathbf{H}$ and (2) use communication-optimal matrix multiplication algorithms to set up the Local Update Computations. The naive approach (Algorithm 2) shares the first property, by parallelizing over rows of $\mathbf{W}$ and columns of $\mathbf{H}$, but it uses parallel matrix multiplication algorithms that communicate more data than necessary. The central intuition for communication-efficient parallel algorithms for computing $\mathbf{HH}^T$, $\mathbf{AH}^T$, $\mathbf{W}^T\mathbf{W}$, and $\mathbf{W}^T\mathbf{A}$ comes from a classification proposed by Demmel et al. [38]. They consider three cases, depending on the relative sizes of the dimensions of the matrices and the number of processors; the four multiplies for NMF fall into either the "one large dimension" or "two large dimensions" cases. MPI-FAUN uses a careful data distribution in order to use a communication-optimal algorithm for each of the matrix multiplications, while at the same time exploiting the parallelism in the LUC.

The algorithm uses a 2D distribution of the data matrix $\mathbf{A}$ across a $p_r \times p_c$ grid of processors (with $p = p_r p_c$), as shown in Figure 2. As we derive in the subsequent subsections, Algorithm 3 performs an alternating method in parallel with a per-iteration bandwidth cost of $O\left(\min\left\{\sqrt{mnk^2/p},nk\right\}\right)$ words, latency cost of $O(\log p)$ messages, and load-balanced computation (up to the sparsity pattern of $\mathbf{A}$ and convergence rates of local BPP computations).

The main improvement of MPI-FAUN over **Naive** involves the computation of $\mathbf{AH}^T$ and $\mathbf{W}^T\mathbf{A}$. By using a 2D distribution of the data matrix, no processor needs access to *all* of one factor matrix, as in the case of **Naive**, where each processor must access either all m rows of $\mathbf{W}$ or all n columns of $\mathbf{H}$. Instead, with MPI-FAUN, each processor must access only m/p_r of the rows of $\mathbf{W}$ and n/p_c of the columns of $\mathbf{H}$, so the number of rows decreases as p increases. This implies the communication cost is reduced, as verified empirically in Figure 7 (the extreme cases correspond to 1D distributions).

To minimize the communication cost and local memory requirements, in the typical case p_r and p_c are chosen so that $m/p_r \approx n/p_c \approx \sqrt{mn/p}$, in which case the bandwidth cost is $O\left(\sqrt{mnk^2/p}\right)$. If the matrix is very tall and skinny, i.e., $m/p > n$, then we choose $p_r = p$ and $p_c = 1$. In this case, the distribution of the data matrix is 1D, and the bandwidth cost is $O(nk)$ words.

The matrix distributions for Algorithm 3 are given in Figure 2; we use a 2D distribution of $\mathbf{A}$ and 1D distributions of $\mathbf{W}$ and $\mathbf{H}$. Recall from Table 2 that $\mathbf{M}_i$ and $\mathbf{M}^i$ denote row and column blocks of $\mathbf{M}$, respectively. Thus, the notation $(\mathbf{W}_i)_j$ denotes the jth row block within the ith row block of $\mathbf{W}$. Lines 3–8 compute

Algorithm 3 $[\mathbf{W},\mathbf{H}] = \text{MPI-FAUN}(\mathbf{A},k)$

Require: $\mathbf{A}$ is an $m \times n$ matrix distributed across a $p_r \times p_c$ grid of processors, k is rank of approximation
Require: Local matrices: $\mathbf{A}_{ij}$ is $m/p_r \times n/p_c$, $\mathbf{W}_i$ is $m/p_r \times k$, $(\mathbf{W}_i)_j$ is $m/p \times k$, $\mathbf{H}_j$ is $k \times n/p_c$, and $(\mathbf{H}_j)_i$ is $k \times n/p$
1: p_{ij} initializes $(\mathbf{H}_j)_i$
2: **while** stopping criteria not satisfied **do**
  /* Compute W given H */
3: p_{ij} computes $\mathbf{U}_{ij} = (\mathbf{H}_j)_i (\mathbf{H}_j)_i^T$
4: compute $\mathbf{HH}^T = \sum_{i,j} \mathbf{U}_{ij}$ using all-reduce across all procs ▷ $\mathbf{HH}^T$ is $k \times k$ and symmetric
5: p_{ij} collects $\mathbf{H}_j$ using all-gather across proc columns
6: p_{ij} computes $\mathbf{V}_{ij} = \mathbf{A}_{ij}\mathbf{H}_j^T$ ▷ $\mathbf{V}_{ij}$ is $m/p_r \times k$
7: compute $(\mathbf{AH}^T)_i = \sum_j \mathbf{V}_{ij}$ using reduce-scatter across proc row to achieve row-wise distribution of $(\mathbf{AH}^T)_i$ ▷ p_{ij} owns $m/p \times k$ submatrix $((\mathbf{AH}^T)_i)_j$
8: p_{ij} computes $(\mathbf{W}_i)_j \leftarrow \text{UpdateW}\ (\mathbf{HH}^T, ((\mathbf{AH}^T)_i)_j)$
  /* Compute H given W */
9: p_{ij} computes $\mathbf{X}_{ij} = (\mathbf{W}_i)_j^T (\mathbf{W}_i)_j$
10: compute $\mathbf{W}^T\mathbf{W} = \sum_{i,j} \mathbf{X}_{ij}$ using all-reduce across all procs ▷ $\mathbf{W}^T\mathbf{W}$ is $k \times k$ and symmetric
11: p_{ij} collects $\mathbf{W}_i$ using all-gather across proc rows
12: p_{ij} computes $\mathbf{Y}_{ij} = \mathbf{W}_i^T \mathbf{A}_{ij}$ ▷ $\mathbf{Y}_{ij}$ is $k \times n/p_c$
13: compute $(\mathbf{W}^T\mathbf{A})^j = \sum_i \mathbf{Y}_{ij}$ using reduce-scatter across proc columns to achieve column-wise distribution of $(\mathbf{W}^T\mathbf{A})^j$ ▷ p_{ij} owns $k \times n/p$ submatrix $((\mathbf{W}^T\mathbf{A})^j)^i$
14: p_{ij} computes $((\mathbf{H}^j)^i)^T \leftarrow \text{UpdateH}\ (\mathbf{W}^T\mathbf{W}, (((\mathbf{W}^T\mathbf{A})^j)^i)^T)$
15: **end while**
Ensure: $\mathbf{W},\mathbf{H} \approx \underset{\tilde{\mathbf{W}} \geqslant 0, \tilde{\mathbf{H}} \geqslant 0}{\text{argmin}} \|\mathbf{A} - \tilde{\mathbf{W}}\tilde{\mathbf{H}}\|$
Ensure: $\mathbf{W}$ is an $m \times k$ matrix distributed row-wise across processors, $\mathbf{H}$ is a $k \times n$ matrix distributed column-wise across processors

$\mathbf{W}$ for a fixed $\mathbf{H}$, and lines 9–14 compute $\mathbf{H}$ for a fixed $\mathbf{W}$; note that the computations and communication patterns for the two alternating iterations are analogous.

In the rest of this section, we derive the per-iteration computation and communication costs, as well as the local memory requirements. We also argue the communication-optimality of the algorithm in the dense case. Table 3 summarizes the results of this section and compares them to Naive-Parallel-AUNMF.

5.2.1 Computation Cost

Local matrix computations occur at lines 3, 6, 9, and 12. In the case that $\mathbf{A}$ is dense, each processor performs

$$\frac{n}{p}k^2 + 2\frac{m}{p_r}\frac{n}{p_c}k + \frac{m}{p}k^2 + 2\frac{m}{p_r}\frac{n}{p_c}k = 4\frac{mnk}{p} + \frac{(m+n)k^2}{p}$$

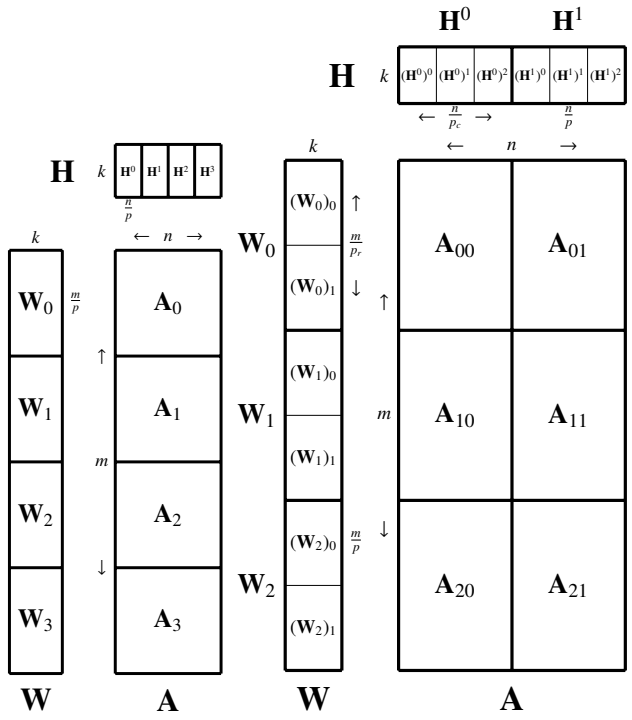


Fig. 2: Data distributions for MPI-FAUN. 1D Distribution on left with $p = p_r = 4$ and $p_c = 1$. 2D Distribution on right with $p_r = 3$ and $p_c = 2$ Note that for the 2D distribution, $\mathbf{A}_{ij}$ is $m/p_r \times m/p_c$, $\mathbf{W}_i$ is $m/p_r \times k$, $(\mathbf{W}_i)_j$ is $m/p \times k$, $\mathbf{H}_j$ is $k \times n/p_c$, and $(\mathbf{H}^j)^i$ is $k \times n/p$.

flops. Recall that the second term on the right hand side has a constant factor of 1 instead of 2 because the local Gram computations (lines 3 and 9) exploit symmetry. In the case that $\mathbf{A}$ is sparse, processor (i,j) performs $(m+n)k^2/p$ flops in computing $\mathbf{U}_{ij}$ and $\mathbf{X}_{ij}$, and $4\text{nnz}(\mathbf{A}_{ij})k$ flops in computing $\mathbf{V}_{ij}$ and $\mathbf{Y}_{ij}$. Local update computations occur at lines 8 and 14. In each case, the symmetric positive semi-definite matrix is $k \times k$ and the number of columns/rows of length k to be computed are m/p and n/p, respectively. These costs together are given by $F(m/p, n/p, k)$. There are computation costs associated with the all-reduce and reduce-scatter collectives (see Section 2.3), both those contribute only to lower order terms: $O(k^2 + mk/p_r + nk/p_c)$.

5.2.2 Communication Cost

Communication occurs during six collective operations (lines 4, 5, 7, 10, 11, and 13). We use the cost expressions presented in Section 2.3 for these collectives. The communication cost of the all-reduces (lines 4 and 10) is $\alpha \cdot 4\log p + \beta \cdot 2k^2$; the cost of the two all-gathers (lines 5 and 11) is $\alpha \cdot \log p + \beta \cdot ((p_r - 1)nk/p + (p_c - 1)mk/p)$; and the cost of the two reduce-scatters (lines 7 and 13) is $\alpha \cdot \log p + \beta \cdot ((p_c - 1)mk/p + (p_r - 1)nk/p)$.

We note that LUC may introduce significant communication cost, depending on the NMF algorithm used. The normalization of columns of $\mathbf{W}$ within **HALS**, for example, introduces an extra $k \log p$ latency cost. We will ignore such costs in our general analysis.

In the case that $m/p < n$, we choose $p_r = \sqrt{mp/n} > 1$ and $p_c = \sqrt{np/m} > 1$, and these communication costs simplify to $\alpha \cdot O(\log p) + \beta \cdot O(mk/p_r + nk/p_c + k^2) = \alpha \cdot O(\log p) + \beta \cdot O(\sqrt{mnk^2/p} + k^2)$. In the case that $m/p \geqslant n$, we choose $p_c = 1$, and the costs simplify to $\alpha \cdot O(\log p) + \beta \cdot O(nk)$.

5.2.3 Memory Requirements

The local memory requirement includes storing each processor's part of matrices $\mathbf{A}$, $\mathbf{W}$, and $\mathbf{H}$. In the case of dense $\mathbf{A}$, this is $mn/p + (m+n)k/p$ words; in the sparse case, processor (i,j) requires $\text{nnz}(\mathbf{A}_{ij})$ words for the input matrix and $(m+n)k/p$ words for the output factor matrices. Local memory is also required for storing temporary matrices $\mathbf{W}_j$, $\mathbf{H}_i$, $\mathbf{V}_{ij}$, and $\mathbf{Y}_{ij}$, of size $2mk/p_r + 2nk/p_c$ words.

In the dense case, assuming $k < n/p_c$ and $k < m/p_r$, the local memory requirement is no more than a constant times the size of the original data. For the optimal choices of p_r and p_c, this assumption simplifies to $k < \max\left\{\sqrt{mn/p}, m/p\right\}$. Note that the second argument of the max applies when the optimal distribution is 1D ($p_r = p$).

We note that if the temporary memory requirements become prohibitive, the computation of $((\mathbf{A}\mathbf{H}^T)_i)_j$ and $((\mathbf{W}^T\mathbf{A})_j)_i$ via all-gathers and reduce-scatters can be blocked, decreasing the local memory requirements at the expense of greater latency costs. When $\mathbf{A}$ is sparse and k is large enough, the memory footprint of the factor matrices can be larger than the input matrix. In this case, the extra temporary memory requirements can become prohibitive; we observed this for a sparse data set with very large dimensions (see Section 6.3.5). We leave the implementation of the blocked algorithm to future work.

5.2.4 Communication Optimality

In the case that $\mathbf{A}$ is dense, Algorithm 3 provably minimizes communication costs. Theorem 5.1 establishes the bandwidth cost lower bound for any algorithm that computes $\mathbf{W}^T\mathbf{A}$ or $\mathbf{A}\mathbf{H}^T$ each iteration. A latency lower bound of $\Omega(\log p)$ exists in our communication model for any algorithm that aggregates global information [19]. For NMF, this global aggregation is necessary in each iteration, for example, in order to compute residual error in the case that $\mathbf{A}$ is distributed across all p processors, because all processors have data that must be accumulated into the global error. Based on the costs derived above, MPI-FAUN is communication optimal under the assumption $k < \sqrt{mn/p}$, matching these lower bounds to within constant factors.

Theorem 5.1 ([38]). Let $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{W} \in \mathbb{R}^{m \times k}$, and $\mathbf{H} \in \mathbb{R}^{k \times n}$ be dense matrices, with $k < n \leqslant m$. If $k < \sqrt{mn/p}$, then any distributed-memory parallel algorithm on p processors that load balances the matrix distributions and computes $\mathbf{W}^T\mathbf{A}$ and/or $\mathbf{A}\mathbf{H}^T$ must communicate at least $\Omega(\min\{\sqrt{mnk^2/p}, nk\})$ words along its critical path.

Proof The proof follows directly from [38, Section II.B]. Each matrix multiplication $\mathbf{W}^T\mathbf{A}$ and $\mathbf{A}\mathbf{H}^T$ has dimensions $k < n \leqslant m$, so the assumption $k < \sqrt{mn/p}$ ensures that neither multiplication has "3 large dimensions." Thus, the communication lower bound is either $\Omega(\sqrt{mnk^2/p})$ in the case of $p > m/n$ (or "2 large dimensions"), or $\Omega(nk)$, in the case of $p < m/n$ (or "1 large dimension"). If $p < m/n$, then $nk < \sqrt{mnk^2/p}$, so the lower bound can be written as $\Omega(\min\{\sqrt{mnk^2/p}, nk\})$.

We note that the communication costs of Algorithm 3 are the same for dense and sparse data matrices (the data matrix itself is never communicated). In the case that $\mathbf{A}$ is sparse, this communication lower bound does not necessarily apply, as the required data movement depends on the sparsity pattern of $\mathbf{A}$. Thus, we cannot

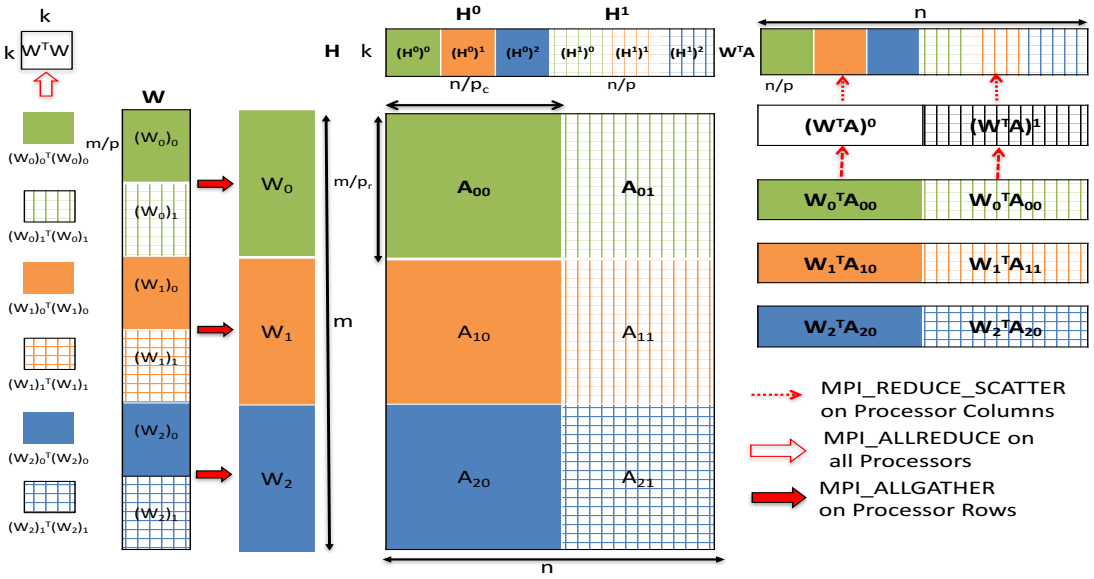


Fig. 3: Parallel matrix multiplications within MPI-FAUN for finding $\mathbf{H}$ given $\mathbf{W}$, with $p_r = 3$ and $p_c = 2$. The computation of $\mathbf{W}^T\mathbf{W}$ appears on the far left; the rest of the figure depicts computation of $\mathbf{W}^T\mathbf{A}$.

make claims of optimality in the sparse case (for general $\mathbf{A}$). The communication lower bounds for $\mathbf{W}^T\mathbf{A}$ and/or $\mathbf{A}\mathbf{H}^T$ (where $\mathbf{A}$ is sparse) can be expressed in terms of hypergraphs that encode the sparsity structure of $\mathbf{A}$ [39]. Indeed, hypergraph partitioners have been used to reduce communication and achieve load balance for a similar problem: computing a low-rank representation of a sparse tensor (without non-negativity constraints on the factors) [40].

6 Experiments

In this section, we describe our implementation of MPI-FAUN and evaluate its performance. We identify a few synthetic and real world data sets to experiment with MPI-FAUN with dimensions that span from hundreds to millions. We compare the performance and exploring scaling behavior of different NMF algorithms – **MU**, **HALS**, and ANLS/BPP (**ABPP**), implemented using the parallel MPI-FAUN framework. The code and the datasets used for conducting the experiments can be downloaded from https://github.com/ramkikannan/nmflibrary.

6.1 Experimental Setup

6.1.1 Data Sets

We used sparse and dense matrices that are either synthetically generated or from real world applications. We explain the data sets in this section.

- Dense Synthetic Matrix: We generate a low rank matrix as the product of two uniform random matrices of size 207,360 × 100 and 100 × 138,240. The dimensions of this matrix are chosen to be evenly divisible for a particular set of processor grids.
- Sparse Synthetic Matrix: We generate a random sparse Erdős-Rényi matrix of the size 207,360 × 138,240 with density of 0.001. That is, every entry is nonzero with probability 0.001.
- Dense Real World Matrix (*Video*): NMF is used on video data for background subtraction in order to detect moving objects. The low rank matrix $\hat{\mathbf{A}} = \mathbf{WH}$ represents background and the error matrix $\mathbf{A} - \hat{\mathbf{A}}$ represents moving objects. In the case of detecting moving objects in streaming videos, the last several minutes of video is taken from the live video camera to construct the non-negative matrix. An algorithm to incrementally adjust the NMF based on the streaming video is presented in [10]. To simulate this scenario, we collected a video in a busy intersection of the Georgia Tech campus at 20 frames per second. From this video, we took video for approximately 12 minutes and then reshaped the matrix such that every RGB frame is a column of our matrix, so that the matrix is dense with size 1,013,400 × 13,824.
- Sparse Real World Matrix (*Webbase*): This data set is a directed sparse graph whose nodes correspond to webpages (URLs) and edges correspond to hyperlinks from one webpage to another. The NMF output of this directed graph helps us understand clusters in graphs. We consider two versions of the data set: *webbase-1M* and *webbase-2001*. The dataset webbase-1M contains about 1 million nodes (1,000,005) and 3.1 million edges (3,105,536), and was first reported by Williams et al. [41]. The version webbase-2001 has about 118 million nodes (118,142,155) and over 1 billion edges (1,019,903,190); it was first reported by Boldi and Vigna [42]. Both data sets are available in the University of Florida Sparse Matrix Collection [43] and the latter *webbase-2001* being the largest among the entire collection.
- Text data (*Stack Exchange*): Stack Exchange is a network of question-and-answer websites on topics in varied fields, each site covering a specific topic, where questions, answers, and users are subject to a reputation award process. There are many Stack Exchange forums, such as *ask ubuntu, mathematics, latex*. We downloaded the latest anonymized dump of all user-contributed content on the Stack Exchange network from [44]. We used only the questions from the most popular site called Stackoverflow and did not include the answers and comments. We removed the standard 571 English stop words (such as *are, am, be, above, below*) and then used snowball stemming available through the Natural Language Toolkit (NLTK) package [45]. After this initial pre-processing, we deleted HTML tags (such as *lt, gt, em*) from the posts. The resulting bag-of-words matrix has a vocabulary of size 627,047 over 11,708,841 documents with 365,168,945 non-zero entries. In this data, the vo-

cabulary is larger than the typical set of English words because it includes variables, constants, and other programming constructs of various programming languages from the user questions.

The size of all the real world data sets were adjusted to the nearest size for uniformly distributing the matrix.

6.1.2 Implementation Platform

We conducted our experiments on "Rhea" at the Oak Ridge Leadership Computing Facility (OLCF). Rhea is a commodity-type Linux cluster with a total of 512 nodes and a 4X FDR Infiniband interconnect. Each node contains dual-socket 8-core Intel Sandy Bridge-EP processors and 128 GB of memory. Each socket has a shared 20MB L3 cache, and each core has a private 256K L2 cache.

Our objective of the implementation is using open source software as much as possible to promote reproducibility and reuse of our code. The entire C++ code is developed using the matrix library Armadillo [46]. In Armadillo, the elements of the dense matrix are stored in column major order and the sparse matrices in Compressed Sparse Column (CSC) format. For dense BLAS and LAPACK operations, we linked Armadillo with Intel MKL – the default LAPACK/BLAS library in RHEA. It is also easy to link Armadillo with OpenBLAS [47]. We use Armadillo's own implementation of sparse matrix-dense matrix multiplication, the default GNU C++ Compiler (g++ (GCC) 4.8.2) and MPI library (Open MPI 1.8.4) on RHEA. We chose the commodity cluster with open source software so that the numbers presented here are representative of common use.

6.1.3 Algorithms

In our experiments, we considered the following algorithms:

- **MU**: MPI-FAUN (Algorithm 3) with MU (Equation (3))
- **HALS**: MPI-FAUN (Algorithm 3) with HALS (Equation (5))
- **ABPP**: MPI-FAUN (Algorithm 3) with BPP (Section 4.3)
- **Naive**: Naive-Parallel-AUNMF (Algorithm 2, Section 5.1)

Our implementation of **Naive** (Algorithm 2) uses BPP but can be easily to extended to **MU**, **HALS**, and other NMF algorithms.

For the algorithms based on MPI-FAUN, we use the processor grid that is closest to the theoretical optimum (see Section 5.2.2) in order to minimize communication costs. See Section 6.3.4 for an empirical evaluation of varying processor grids for a particular algorithm and data set.

To ensure fair comparison among algorithms, the same random seed is used across different methods appropriately. That is, the initial random matrix $\mathbf{H}$ is generated with the same random seed when testing with different algorithms (note that $\mathbf{W}$ need not be initialized). In our experiments, we use number of iterations as the stopping criteria for all the algorithms.

While we would like to compare against other high-performance NMF algorithms in the literature, the only other distributed-memory implementations of which we're aware are implemented using Hadoop and are designed only for sparse matrices [24], [27], [13], [26] and [25]. We stress that Hadoop is not designed for high performance computing of iterative numerical algorithms, requiring disk I/O between steps, so a run time comparison between a Hadoop implementation and a C++/MPI implementation is not a fair comparison of parallel algorithms. A qualitative example of differences in run time is that a Hadoop implementation of the MU algorithm on a large sparse matrix of size $2^{17} \times 2^{16}$ with 2×10^8 nonzeros (with k=8) takes on the order of 50 minutes per iteration [27], while our MU implementation takes 0.065 seconds per iteration for the synthetic data set (which is an order of magnitude larger in terms of rows, columns, and nonzeros) running on only 16 nodes.

6.2 Relative Error over Time

There are various metrics to compare the quality of the NMF algorithms [10]. The most common among these metrics are (a) relative error and (b) projected gradient. The former represents the closeness of the low rank approximation $\hat{\mathbf{A}} \approx \mathbf{WH}$, which is generally the optimization objective. The latter represent the quality of the produced low rank factors and the stationarity of the final solution. These metrics are also used as the stopping criterion for terminating the iteration of the NMF algorithm as in line 2 of Algorithm 1. Typically a combination of the number of iterations along with improvement of these metrics until a tolerance is met is used as stopping criterion. In this paper, we use relative error for the comparison as it is monotonically decreasing, as opposed to projected gradient of the low rank factors, which shows oscillations over iterations. The relative error can be formally defined as $\|\mathbf{A}-\mathbf{WH}\|_F/\|\mathbf{A}\|_F$.

In Figure 4, we measure the relative error at the end of every iteration (i.e., after the updates of both $\mathbf{W}$ and $\mathbf{H}$) for all three algorithms **MU**, **HALS**, and **ABPP**, and we plot the relative error over time (each mark represents an iteration). We consider three real world datasets, *video*, *stack exchange* and *webbase-1M*, and set $k=50$. We used only the number of iterations as stopping criterion and, just for this section, ran all the algorithms for 30 iterations. We note that the convergence behavior and computed factors can vary over different initializations; we used the same initial values across all three algorithms in these experiments. Also, we observed that for these data sets, the convergence behavior was not sensitive to initialization (the final residual errors varied by less than 1% in our experiments). NMF solutions are guaranteed to be unique in certain cases, with mild assumptions on the input data [48], [49], but we do not check those assumptions for these datasets.

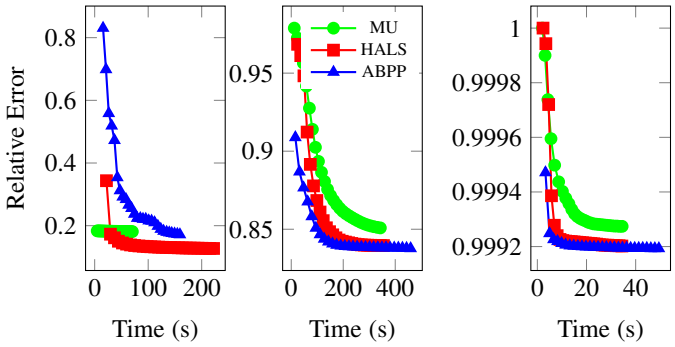
To begin with, we explain the observations on the dense *video* dataset presented in Figure 4a. The relative error of **MU** is highest at 0.1812 after 30 iterations and **HALS**'s is the least with 0.1273. **ABPP**'s relative error was 0.1716 and if ran longer **ABPP** would have converged similar to **HALS**.

We can observe that the relative error of *stack exchange* from Figure 4b is better than *webbase-1M* from Figure 4c over all three algorithms. In the case of the *stack exchange* dataset, the relative errors after 30 iterations follow the pattern **MU** > **HALS** > **ABPP**, with values 0.8509, 0.8395, and 0.8377 respectively. However, the difference in relative error for the *webbase-1M* dataset is negligible, though the relative ordering of **MU** > **HALS** > **ABPP** is consistent, with values of 0.99927 for **MU** 0.99920 for **HALS** and 0.99919 for **ABPP**.

In general, for these datasets **ABPP** identifies better approximations and converges faster than **MU** and **HALS** despite the extra per-iteration time, which is consistent with the literature [10], [12]. However, for the sparse datasets, the differences in relative error are small across the NMF algorithms.

6.3 Time Per Iteration

In this section we focus on per-iteration time of all the algorithms. We report four types of experiments, varying the number of processors (Section 6.3.2), the rank of the approximation (Section 6.3.3), the shape of the processor grid (Section 6.3.4), and scaling



(a) Video (b) Stack Exchange (c) Webbase

Fig. 4: Relative error comparison of **MU**, **HALS**, **ABPP** on real world datasets.

up the dataset size. For each experiment we report a time breakdown in terms of the overall computation and communication steps (described in Section 6.3.1) shared by all algorithms.

6.3.1 Time Breakdown

To differentiate the computation and communication costs among the algorithms, we present the time breakdown among the various tasks within the algorithms for all performance experiments. For Algorithm 3, there are three local computation tasks and three communication tasks to compute each of the factor matrices:

- **MM**, computing a matrix multiplication with the local data matrix and one of the factor matrices;
- **LUC**, local updates either using **ABPP** or applying the remaining work of the **MU** or **HALS** updates (i.e., the total time for both $UpdateW$ and $UpdateH$ functions);
- **Gram**, computing the local contribution to the Gram matrix;
- **All-Gather**, to compute the global matrix multiplication;
- **Reduce-Scatter**, to compute the global matrix multiplication;
- **All-Reduce**, to compute the global Gram matrix.

In our results, we do not distinguish the costs of these tasks for **W** and **H** separately; we report their sum, though we note that we do not always expect balance between the two contributions for each task. Algorithm 2 performs all of these tasks except Reduce-Scatter and All-Reduce; all of its communication is in All-Gather.

6.3.2 Scaling p: Strong Scaling

Figure 5 presents a strong scaling experiment with four data sets: *sparse synthetic*, *dense synthetic*, *webbase-1M*, and *video*. In this experiment, for each data set and algorithm, we use low rank $k = 50$ and vary the number of processors (with fixed problem size). We use $\{1, 6, 24, 54, 96\}$ nodes; since each node has 16 cores, this corresponds to {16,96,384,864,1536} cores. We report average per-iteration times.

We highlight three main observations from these experiments:

1) **Naive** is slower than all other algorithms for large p;
2) **MU**, **HALS**, and **ABPP** (algorithms based on MPI-FAUN) scale up to over 1000 processors;
3) the relative per-iteration cost of LUC decreases as p increases (for all algorithms), and therefore the extra per-iteration cost of **ABPP** (compared with **MU** and **HALS**) becomes negligible.

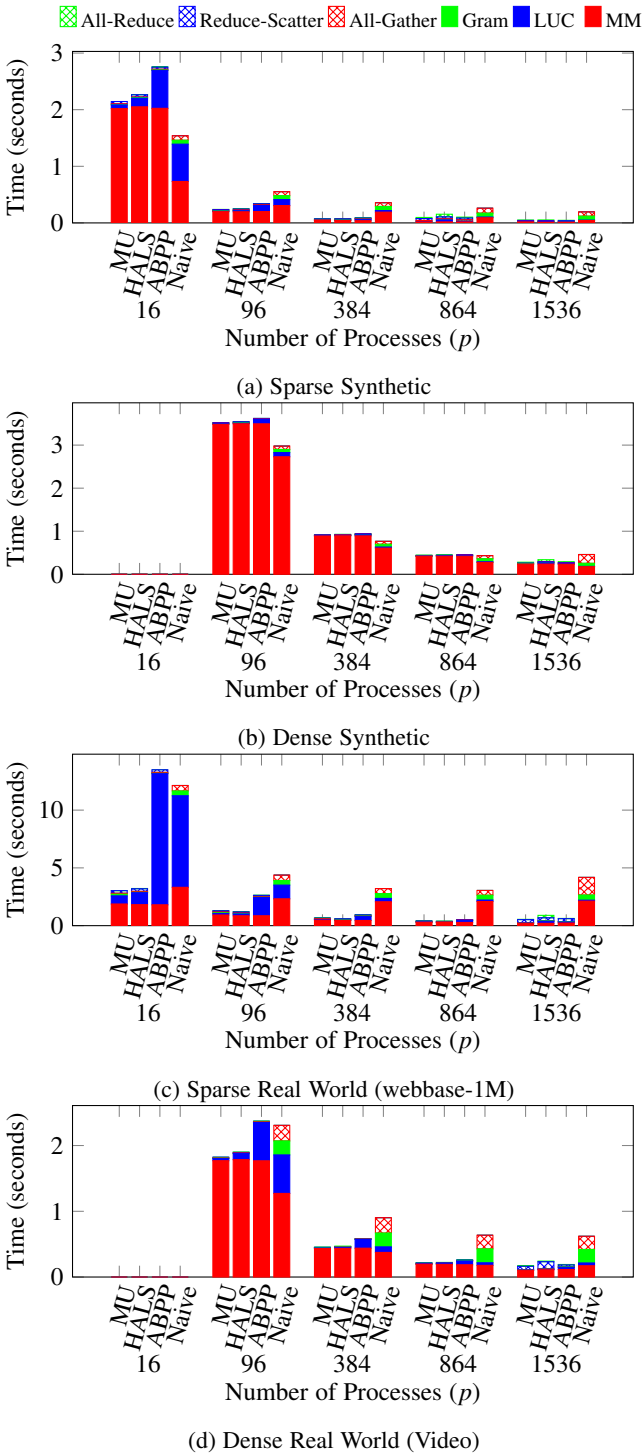
6.3.2.1 Observation 1: For the Sparse Synthetic data set, **Naive** is 4.2× slower than the fastest algorithm (**ABPP**) on 1536 processors; for the Dense Synthetic data set, **Naive** is 1.6× slower than the fastest algorithm (**MU**) at that scale. The slowdown increases to 7.7× and 3.6× for the sparse and dense real-world datasets, respectively. Nearly all of this slowdown is due to the communication costs of **Naive**. Theoretical and practical evidence supporting the first observation is also reported in our previous paper [17]. However, we also note that **Naive** is the fastest algorithm for the smallest p for each problem, which is largely due to reduced MM time. Each algorithm performs exactly the same number of flops per MM; the efficiency of **Naive** for small p is due to cache effects. For example, for the Dense Synthetic problem on 96 processors, the output matrix of **Naive**'s MM fits in L2 cache, but the output matrix of MPI-FAUN's MM does not; these effects disappear as p increases.

6.3.2.2 Observation 2: Algorithms based on MPI-FAUN (**MU**, **HALS**, **ABPP**) scale well, up to over 1000 processors. All algorithms' run times decrease as p increases, with the exception of the Sparse Real World data set, in which case all algorithms slow down scaling from $p = 864$ to $p = 1536$ (we attribute this lack of scaling to load imbalance). For sparse problems, comparing $p = 16$ to $p = 1536$ (a factor increase of 96), we observe speedups from **ABPP** of 59× (synthetic) and 22× (real world). For dense problems, comparing $p = 96$ to $p = 1536$ (a factor increase of 16), **ABPP**'s speedup is 12× for both problems. **MU** and **HALS** demonstrate similar scaling results. For comparison, speedups for **Naive** were 8× and 3× (sparse) and 6× and 4× (dense).

6.3.2.3 Observation 3: **MU**, **HALS**, and **ABPP** share all the same subroutines except those that are characterized as LUC. Considering only LUC subroutines, **MU** and **HALS** require fewer operations than **ABPP**. However, **HALS** has to make one additional communication for normalization of **W**. For small p, these cost differences are apparent in Figure 5. For example, for the sparse real world data set on 16 processors, **ABPP**'s LUC time is 16× that of **MU**, and the per iteration time differs by a factor of 4.5. However, as p increases, the relative time spent in LUC decreases, so the extra time taken by **ABPP** has less of an effect on the total per iteration time. By contrast, for the dense real world data set on 1536 processors, **ABPP** spends a factor of 27 times more time in LUC than **MU** but only 11% longer over the entire iteration. For the synthetic data sets, LUC takes 24% (sparse) on 16 processors and 84% (dense) on 96 processors, and that percentage drops to 11% (sparse) and 15% (dense) on 1536 processors.

These trends can also be seen theoretically (Table 3). We expect local computations like MM, LUC, and Gram to scale like $1/p$, assuming load balance is preserved. If communication costs are dominated by the number of words being communicated (i.e., the communication is bandwidth bound), then we expect time spent in communication to scale like $1/\sqrt{p}$, and at least for dense problems, this scaling is the best possible. Thus, communication costs will eventually dominate computation costs for all NMF problems, for sufficiently large p. (Note that if communication is latency bound and proportional to the number of messages, then time spent communicating actually increases with p.)

The overall conclusion from this empirical and theoretical observation is that the extra per-iteration cost of **ABPP** over alternatives like **MU** and **HALS** decreases as the number of processors p increases. As shown in Section 6.2 the faster error reduction of **ABPP** typically reduces the overall time to solution compared with the alternatives even it requires more time for



(a) Sparse Synthetic

(b) Dense Synthetic

(c) Sparse Real World (webbase-1M)

(d) Dense Real World (Video)

Fig. 5: Per-iteration times with $k=50$, varying p (strong scaling).

each iteration. Our conclusion is that as we scale up p, this tradeoff is further relaxed so that **ABPP** becomes more and more advantageous for both quality and performance.

6.3.3 Scaling k

Figure 6 presents an experiment scaling up the low rank value k from 10 to 50 with each of the four data sets. In this experiment, for each data set and algorithm, the problem size is fixed and the number of processors is fixed to $p=864$. As in Section 6.3.2, we report the average per-iteration times.

We highlight two observations from these experiments:

1) **Naive** is plagued by communication time that increases linearly with k;
2) **ABPP**'s time increases more quickly with k than those of **MU** or **HALS**;

6.3.3.1 Observation 1: We see from the synthetic data sets (Figures 6a and 6b) that the overall time of **Naive** increases more rapidly with k than any other algorithm and that the increase in time is due mainly to communication (All-Gather). Table 3 predicts that **Naive** communication volume scales linearly with k, and we see that in practice the prediction is almost perfect with the synthetic problems. This confirms that the communication is dominated by bandwidth costs and not latency costs (which are constant with respect to k). We note that the communication cost of MPI-FAUN scales like $\sqrt{k}$, which is why we don't see as dramatic an increase in communication time for **MU**, **HALS**, or **ABPP** in Figure 6.

6.3.3.2 Observation 2: Focusing attention on time spent in LUC computations, we can compare how **MU**, **HALS**, and **ABPP** scale differently with k. We see a more rapid increase of LUC time for **ABPP** than **MU** or **HALS**; this is expected because the LUC computations unique to **ABPP** require between $O(k^3)$ and $O(k^4)$ operations (depending on the data) while the unique LUC computations for **MU** and **HALS** are $O(k^2)$, with all other parameters fixed. Thus, the extra per-iteration cost of **ABPP** increases with k, so the advantage of **ABPP** of better error reduction must also increase with k for it to remain superior at large values of k. We also note that although the number of operations within MM grows linearly with k, we do not observe much increase in time from $k=10$ to $k=50$; this is due to the improved efficiency of local MM for larger values of k.

6.3.4 Varying Processor Grid

In this section we demonstrate the effect of the dimensions of the processor grid on per-iteration performance. For a fixed total number of processors p, the communication cost of Algorithm 3 varies with the choice of p_r and p_c. To minimize the amount of data communicated, the theoretical analysis suggests that the processor grid should be chosen to make the sizes of the local data matrix as square as possible. This implies that if $m/p > n$, $p_r = p$ and $p_c = 1$ is the optimal choice (a 1D processor grid); likewise if $n/p > m$ then a 1D processor grid with $p_r = 1$ and $p_c = p$ is the optimal choice. Otherwise, a 2D processor grid minimizes communication with $p_r \approx \sqrt{mp/n}$ and $p_c \approx \sqrt{np/m}$ (subject to integrality and $p_r p_c = p$).

Figure 7 presents a benchmark of **ABPP** for the Sparse Synthetic data set for fixed values of p and k. We vary the processor grid dimensions from both 1D grids to the 2D grid that matches the theoretical optimum exactly. Because the sizes of the Sparse Synthetic matrix are $172{,}800 \times 115{,}200$ and the number of processors is 1536, the theoretically optimal grid is $p_r = \sqrt{mp/n} = 48$ and $p_c = \sqrt{np/m} = 32$. The experimental results confirm that this processor grid is optimal, and we see that the time spent communicating increases as the processor grid deviates from the optimum, with the 1D grids performing the worst.

6.3.5 Scaling up to Very Large Sparse Datasets

In this section, we test MPI-FAUN by scaling up the problem size. While we've used *webbase-1M* in previous experiments, we consider *webbase-2001* in this section as it is the largest sparse data in University of Florida Sparse Matrix Collection [43]. The former dataset has about 1 million nodes and 3 million edges,

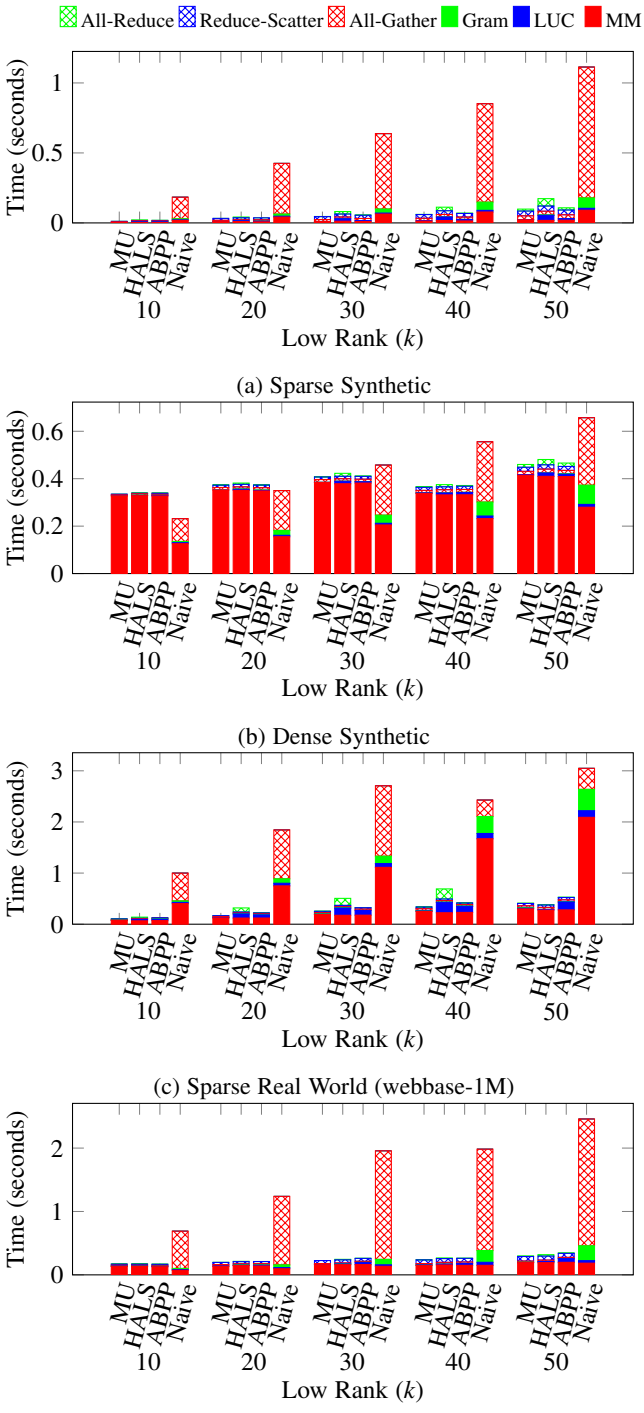


Fig. 6: Per-iteration times with $p = 864$, varying low rank k.

whereas the latter dataset has over 100 million nodes and 1 billion edges (see Section 6.1.1 for more details). Not only is the size of the input matrix increased by two orders of magnitude (because of the increase in the number of edges), but also the size of the output matrices is increased by two orders of magnitude (because of the increase in the number of nodes).

In fact, with a low rank of $k = 50$, the size of the output matrices dominates that of the input matrix: **W** and **H** together require a total of 88 GB, while **A** (stored in compressed column format) is only 16 GB. At this scale, because each node (consisting of 16

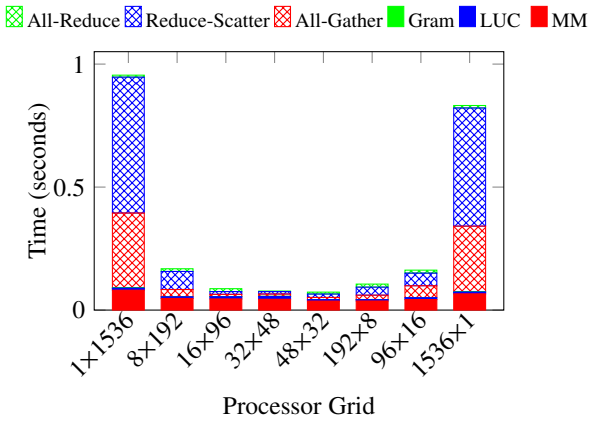


Fig. 7: Tuning processor grid for **ABPP** on Sparse Synthetic data set with $p = 1536$ and $k = 50$.

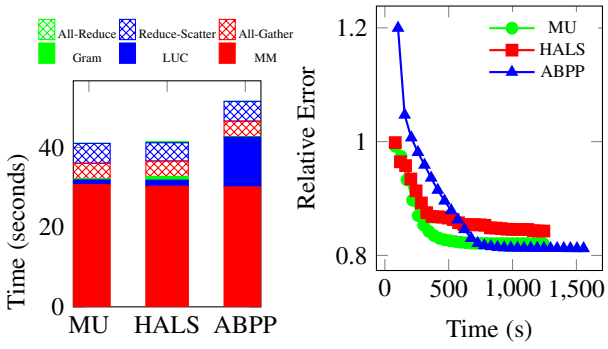
cores) of Rhea has 128 GB of memory, multiple nodes are required to store the input and output matrices with room for other intermediate values. As mentioned in Section 5.2.3, MPI-FAUN requires considerably more temporary memory than necessary when the output matrices require more memory than the input matrix. While we were not limited by this property for the other sparse matrices, the *webbase-2001* matrix dimensions are so large that we need the memories of tens of nodes to run the algorithm. Thus, we report results only for the largest number of processors in our experiments: 1536 processors (96 nodes). The extra temporary memory used by MPI-FAUN is a latency-minimizing optimization; the algorithm can be updated to avoid this extra memory cost using a blocked matrix multiplication algorithm. The extra memory can be reduced to a negligible amount at the expense of more messages between processors and synchronizations across the parallel machine.

We present results for *webbase-2001* in Figure 8. The average per-iteration timing results are consistent with the observations from other synthetic and real world sparse datasets as discussed in Section 6.3.2, though the raw times are about 2 orders of magnitude larger, as expected. In the case of the error plot, as observed in other experiments, **ABPP** achieves smaller error (by 1%) than other algorithms after converging; however **MU** and **HALS** initially outperform **ABPP**. We also see that **MU** outperforms **HALS** in the first 30 iterations. At the 30th iteration, the error for **HALS** is still improving at the third decimal, whereas **MU**'s is improving at the fourth decimal. We suspect that over a greater number of iterations the error of **HALS** could become smaller than that of **MU**, which would be more consistent with other datasets.

6.4 Interpretation of Results

We present results from two of the real world datasets in the Supplemental Material. The first example shows background separation of the *video* data, and the second example shows topic modeling output on the *stack exchange* text dataset. The details of these datasets are presented in Section 6.1.1.

While the literature covers more detail about fine tuning NMF and different NMF variants for higher quality results on these two tasks, our main focus is to show how quickly we can produce baseline NMF solutions. In Figure 1 of the Supplemental Material, we can see the background is removed and the moving objects (e.g., cars) are visible. Similarly, Table 1 of Supplemental Material shows that the NMF solution discriminates among topics and and finds coherent keywords for each topic.



(a) Time (b) Error

Fig. 8: NMF comparison on *webbase-2001* for k=50 on 1536 processors.

7 Conclusion

In this paper, we propose a high-performance distributed-memory parallel framework for NMF algorithms that iteratively update the low rank factors in an alternating fashion. Our parallelization scheme is designed to avoid communication overheads and scales well to over 1500 cores. The framework is flexible, being (a) expressive enough to leverage many different NMF algorithms and (b) efficient for both sparse and dense matrices of sizes that span from a few hundreds to hundreds of millions. Our open-source software implementation is available for download.

For solving data mining problems at today's scale, parallel computation and distributed-memory systems are becoming prerequisites. We argue in this paper that by using techniques from high-performance computing, the computations for NMF can be performed very efficiently. Our framework allows for the HPC techniques (efficient matrix multiplication) to be separated from the data mining techniques (choice of NMF algorithm), and we compare data mining techniques at large scale, in terms of data sizes and number of processors. One conclusion we draw from the empirical and theoretical observations is that the extra per-iteration cost of **ABPP** over alternatives like **MU** and **HALS** decreases as the number of processors p increases, making **ABPP** more advantageous in terms of both quality and performance at larger scales. By reporting time breakdowns that separate local computation from interprocessor communication, we also see that our parallelization scheme prevents communication from bottlenecking the overall computation; our comparison with a naive approach shows that communication can easily dominate the running time of each iteration.

In future work, we would like to extend MPI-FAUN algorithm to dense and sparse tensors, computing the CANDECOMP/PARAFAC decomposition in parallel with non-negativity constraints on the factor matrices. We plan on extending our software to include more NMF algorithms that fit the AU-NMF framework; these can be used for both matrices and tensors. We would also like to explore more intelligent distributions of sparse matrices: while our 2D distribution is based on evenly dividing rows and columns, it does not necessarily load balance the nonzeros of the matrix, which can lead to load imbalance in matrix multiplications. We are interested in using graph and hypergraph partitioning techniques to load balance the memory and computation while at the same time reducing communication costs as much as possible.

8 Acknowledgements

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. OAC-1642385. This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. This project was partially funded by the Laboratory Director's Research and Development fund. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy.

Also, partial funding for this work was provided by AFOSR Grant FA9550-13-1-0100, National Science Foundation (NSF) grants IIS-1348152, ACI-1338745, and ACI-1642385, Defense Advanced Research Projects Agency (DARPA) XDATA program grant FA8750-12-2-0309. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the USDOE, NERSC, AFOSR, NSF or DARPA.

References

[1] D. Seung and L. Lee, "Algorithms for non-negative matrix factorization," *NIPS*, vol. 13, pp. 556–562, 2001.

[2] V. P. Pauca, F. Shahnaz, M. W. Berry, and R. J. Plemmons, "Text mining using nonnegative matrix factorizations," in *Proceedings of SDM*, 2004.

[3] P. O. Hoyer, "Non-negative matrix factorization with sparseness constraints," *JMLR*, vol. 5, pp. 1457–1469, 2004. [Online]. Available: www.jmlr.org/papers/volume5/hoyer04a/hoyer04a.pdf

[4] R. Fujimoto, A. Guin, M. Hunter, H. Park, G. Kanitkar, R. Kannan, M. Milholen, S. Neal, and P. Pecher, "A dynamic data driven application system for vehicle tracking," *Procedia Computer Science*, vol. 29, pp. 1203–1215, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2014.05.108

[5] T. Bouwmans, A. Sobral, S. Javed, S. K. Jung, and E.-H. Zahzah, "Decomposition into low-rank plus additive matrices for background/foreground separation: A review for a comparative evaluation with a large-scale dataset," *arXiv preprint arXiv:1511.01245*, 2015.

[6] H. Kim and H. Park, "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis," *Bioinformatics*, vol. 23, no. 12, pp. 1495–1502, 2007. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btm134

[7] A. Cichocki, R. Zdunek, A. H. Phan, and S.-i. Amari, *Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation*. Wiley, 2009.

[8] D. Kuang, C. Ding, and H. Park, "Symmetric nonnegative matrix factorization for graph clustering," in *Proceedings of SDM*, 2012, pp. 106–117. [Online]. Available: http://epubs.siam.org/doi/pdf/10.1137/1.9781611972825.10

[9] D. Kuang, S. Yun, and H. Park, "SymNMF: nonnegative low-rank approximation of a similarity matrix for graph clustering," *Journal of Global Optimization*, pp. 1–30, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10898-014-0247-2

[10] J. Kim, Y. He, and H. Park, "Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework," *Journal of Global Optimization*, vol. 58, no. 2, pp. 285–319, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10898-013-0035-4

[11] N.-D. Ho, P. V. Dooren, and V. D. Blondel, "Descent methods for nonnegative matrix factorization," *CoRR*, vol. abs/0801.3199, 2008.

[12] J. Kim and H. Park, "Fast nonnegative matrix factorization: An active-set-like method and comparisons," *SIAM Journal on Scientific Computing*, vol. 33, no. 6, pp. 3261–3281, 2011. [Online]. Available: http://dx.doi.org/10.1137/110821172

[13] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the KDD*. ACM, 2011, pp. 69–77. [Online]. Available: http://dx.doi.org/10.1145/2020408.2020426

[14] Y.-X. Wang and Y.-J. Zhang, "Nonnegative matrix factorization: A comprehensive review," *TKDE*, vol. 25, no. 6, pp. 1336–1353, June 2013. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2012.51

[15] Y. Xu and W. Yin, "A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion," *SIAM Journal on Imaging Sciences*, vol. 6, no. 3, pp. 1758–1789, 2013.

[16] K. Huang, N. D. Sidiropoulos, and A. P. Liavas, "A flexible and efficient algorithmic framework for constrained matrix and tensor factorization," *IEEE Transactions on Signal Processing*, vol. 64, no. 19, pp. 5052–5065, Oct 2016.

[17] R. Kannan, G. Ballard, and H. Park, "A high-performance parallel algorithm for nonnegative matrix factorization," in *Proceedings of the 21st ACM SIGPLAN Symposium on PPoPP*, ser. PPoPP '16. New York, NY, USA: ACM, February 2016, pp. 9:1–9:11. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851152

[18] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005. [Online]. Available: http://hpc.sagepub.com/content/19/1/49.abstract

[19] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007. [Online]. Available: http://dx.doi.org/10.1002/cpe.1206

[20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: http://dx.doi.org/10.14778/2212351.2212354

[21] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine Learning in Apache Spark," May 2015. [Online]. Available: http://arxiv.org/abs/1505.06807

[22] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 979–990.

[23] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 975–986, 2014.

[24] R. Liao, Y. Zhang, J. Guan, and S. Zhou, "CloudNMF: A MapReduce implementation of nonnegative matrix factorization for large-scale biological datasets," *Genomics, proteomics & bioinformatics*, vol. 12, no. 1, pp. 48–51, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.gpb.2013.06.001

[25] C. Faloutsos, A. Beutel, E. P. Xing, E. E. Papalexakis, A. Kumar, and P. P. Talukdar, "Flexi-FaCT: Scalable flexible factorization of coupled tensors on Hadoop," in *Proceedings of the SDM*, 2014, pp. 109–117. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611973440.13

[26] J. Yin, L. Gao, and Z. Zhang, "Scalable nonnegative matrix factorization with block-wise updates," in *Machine Learning and Knowledge Discovery in Databases*, ser. LNCS, vol. 8726, 2014, pp. 337–352. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44845-8_22

[27] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang, "Distributed nonnegative matrix factorization for web-scale dyadic data analysis on MapReduce," in *Proceedings of the WWW*. ACM, 2010, pp. 681–690. [Online]. Available: http://dx.doi.org/10.1145/1772690.1772760

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[29] S. Arora, R. Ge, R. Kannan, and A. Moitra, "Computing a nonnegative matrix factorization–provably," in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. ACM, 2012, pp. 145–162.

[30] K. Huang, N. D. Sidiropoulos, and A. Swamiy, "Nmf revisited: New uniqueness results and algorithms," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 4524–4528.

[31] K. Huang, N. D. Sidiropoulos, and A. Swami, "Non-negative matrix factorization revisited: Uniqueness and algorithm for symmetric decomposition," *IEEE Transactions on Signal Processing*, vol. 62, no. 1, pp. 211–224, Jan 2014.

[32] J. P. Fairbanks, R. Kannan, H. Park, and D. A. Bader, "Behavioral clusters in dynamic graphs," *Parallel Computing*, vol. 47, pp. 38–50, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2015.03.002

[33] D. Grove, J. Milthorpe, and O. Tardieu, "Supporting array programming in X10," in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY'14, 2014, pp. 38:38–38:43. [Online]. Available: http://doi.acm.org/10.1145/2627373.2627380

[34] E. Mejía-Roa, D. Tabas-Madrid, J. Setoain, C. García, F. Tirado, and A. Pascual-Montano, "NMF-mGPU: non-negative matrix factorization on multi-GPU systems," *BMC bioinformatics*, vol. 16, no. 1, p. 43, 2015. [Online]. Available: http://dx.doi.org/10.1186/s12859-015-0485-4

[35] D. L. Sun and C. Févotte, "Alternating direction method of multipliers for non-negative matrix factorization with the beta-divergence," in *2014 IEEE ICASSP*, May 2014, pp. 6201–6205.

[36] N. Guan, D. Tao, Z. Luo, and B. Yuan, "Nenmf: An optimal gradient method for nonnegative matrix factorization," *IEEE Transactions on Signal Processing*, vol. 60, no. 6, pp. 2882–2898, June 2012.

[37] A. Cichocki and P. Anh-Huy, "Fast local algorithms for large scale nonnegative matrix and tensor factorizations," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 92, no. 3, pp. 708–721, 2009.

[38] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *Proceedings of IPDPS*, 2013, pp. 261–272. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.80

[39] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Brief announcement: Hypergraph partitioning for parallel sparse matrix-matrix multiplication," in *Proceedings of SPAA*, 2015, pp. 86–88. [Online]. Available: http://doi.acm.org/10.1145/2755573.2755613

[40] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of SC*. ACM, 2015, pp. 77:1–77:11. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807624

[41] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178 – 194, 2009.

[42] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proceedings of the*, ser. WWW '04, New York, NY, USA, 2004, pp. 595–602. [Online]. Available: http://doi.acm.org/10.1145/988672.988752

[43] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[44] S. E. Inc, "Stack exchange," Last Accessed 26-Jun-2017. [Online]. Available: https://archive.org/details/stackexchange

[45] E. Loper and S. Bird, "Nltk: The natural language toolkit," in *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ser. ETMTNLP '02, 2002, pp. 63–70. [Online]. Available: www.nltk.org

[46] C. Sanderson, "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," NICTA, Tech. Rep., 2010. [Online]. Available: http://arma.sourceforge.net/armadillo_nicta_2010.pdf

[47] Z. Xianyi, "Openblas," Last Accessed 03-Dec-2015. [Online]. Available: http://www.openblas.net

[48] A. R. Benson, J. D. Lee, B. Rajwa, and D. F. Gleich, "Scalable methods for nonnegative matrix factorizations of near-separable tall-and-skinny matrices," in *Advances in Neural Information Processing Systems*, 2014, pp. 945–953.

[49] K. Huang, X. Fu, and N. D. Sidiropoulos, "Anchor-free correlated topic modeling: Identifiability and algorithm," in *Advances in Neural Information Processing Systems*, 2016, pp. 1786–1794.

Ramakrishnan Kannan is a Computational Data Scientist in Oak Ridge National Laboratory. He received his Ph.D in Computer Science from College of Computing, Georgia Institute of Technology advised by Prof. Haesun Park. He worked on Data Analytics group at IBM TJ Watson Research Center and was an IBM Master Inventor. He has M.Sc (Engg) from Indian Institute of Science.

Grey Ballard Grey Ballard is an Assistant Professor in the Computer Science Department at Wake Forest University. After receiving his PhD in computer science from the University of California Berkeley in 2013, he was a Truman Fellow at Sandia National Laboratories in Livermore, CA. He received his BS in math and computer science at Wake Forest in 2006 and his MA in math at Wake Forest in 2008. His work has been recognized with the SIAM Linear Algebra Prize and three conference best paper awards, at SPAA, IPDPS, and ICDM.

Haesun Park is a IEEE and SIAM Fellow and Professor in the School of Computational Science and Engineering, Georgia Institute of Technology. She has played major leadership roles as the Executive Director of the Center for Data Analytics, Georgia Tech, general chair for the SIAM Conference on Data Mining, and editorial board member of SIAM and IEEE journals. She received a Ph.D. in Computer Science from Cornell University in 1987 and B.S. in Mathematics from Seoul National University with the University President's Medal.